A METHOD AND SYSTEM FOR HIGHLY-PARALLEL LOGGING AND RECOVERY OPERATION IN MAIN-MEMORY TRANSACTION PROCESSING SYSTEMS

BACKGROUND OF THE INVENTION

Technical Field

This invention relates to main-memory transaction processing systems. More specifically, the present invention relates to a logging method and system for recovery of a main-memory database in a transaction processing system.

Description of Related Art

A transaction processing system must process transactions in such a manner that "consistency" and "durability" of data are maintained even in the event of a failure such as a system crash. Consistency of data is preserved when transactions are performed in an atomic and consistent manner so that the data initially in a consistent state is transformed into another consistent state. Durability of data is preserved when changes made to the data by "committed" transactions (transactions completed successfully) survive system failures. A database often refers to the data on which transactions are performed.

To achieve consistency and durability of a database, most transaction processing systems perform a process called logging, the process of recording updates in terms of log records in a log file. In case of a failure, these log records are used to undo the changes by incomplete transactions, thereby recovering the database into a consistent state before the failure. These log records are also used to redo the changes made by the committed transactions, thereby maintaining the durable database.

Recovering a consistent and durable database after a system crash using only the log records would require a huge volume of log data because all the log records that have been generated since the creation of the database must be saved. Therefore, the process called "checkpointing" is often used where the database is copied to a disk in a regular interval so that only the log records created since the last checkpointing need to be stored. In practice, each page in the database has a "dirty flag" indicating any modification by a transaction so that only the pages modified since the last checkpointing are copied to a disk.

FIG. 1 shows a conventional recovery architecture of a main-memory database

management system (DBMS), where two backups (101 and 102) are maintained with a log (103). A single checkpointing process updates only one of the backups, and successive checkpointing processes alternate between them. To coordinate the alternation, each in-memory database page has two dirty flags. When a transaction modifies a page, it sets both flags indicating modification by the transaction. When a checkpointing process flushes the page to the first backup, the first flag is reset to indicate that no further checkpointing needs be done to the first backup. Similarly, when a successive checkpointing process flushes the page to the second backup, the second flag is reset.

FIG. 2 shows a conventional restart process using a backup of the database and the log in main-memory DBMSs. The restart process comprises the following four steps. First, the most recent backup is read into main memory (BR for "backup read") (201). Second, the database is restored to the one as existed at the time the backup was made (BP for "backup play") (202). Third, the log records are read into main memory (LR for "log read") (203). Fourth, the database is restored to the one of the most recent consistent state using the log records (LP for "log play") (204).

FIG. 3a and 3b are is flow charts of the conventional two-pass log-play process. To restore the database to the one of the most recent consistent state, the log records generated by all the committed transactions need to be played, but the log records generated by so-called "loser transactions" that were active at the time of system crash have to be skipped (A transaction is said to be a loser when there is a matching transaction start log record but no transaction end record). For this purpose, all the log records encountered scanning the log from the checkpointing start log record to the end of the log are played (307). Then, the changes by the log records of the loser transactions are rolled back (308).

To identify loser transactions, a loser transaction table (LTT) is maintained, which has two fields, TID and Last LSN. This table is initialized with the active transactions recorded in the checkpointing start log record (301). When encountering a transaction start record (302), a matching entry is created in the LTT (305). When encountering a transaction end (either commit or abort) record (303), the matching entry is removed from the LTT (306). Otherwise (304), the LSN of the current log record is recorded in the Last LSN field of the matching LTT entry (307). When reaching the end of the log, the transactions that have matching entries still in the LTT are losers. The most recent record of a loser transaction can be located using the Last LSN field of the matching LTT entry, and other records of the transaction can be located by chasing the Last LSN field of accessed log records backward.

When using the physical logging method of the conventional art, log records must be

applied in the order of log-record creation during the LP process. That is, logs created earlier must be used first to redo some of the updates. The conventional logging method imposes the sequential ordering because the undo and redo operations are not commutative and associative. This sequential ordering requirement imposes a lot of constraints in the system design.

In a main-memory DBMS, for example, where the entire database is kept in main memory, disk access for logging acts as a bottleneck in the system performance. In order to reduce such a bottleneck, employment of multiple log disks may be conceived to distribute the processing. The use of multiple log disks, however, is not easily amenable to the conventional logging method because there is necessarily an overhead of merging log records in the order of creation during the step of LP.

Therefore, there is a need for an efficient logging system that may comport with massive parallel operations in distributed processing.

OBJECTS AND SUMMARY OF THE INVENTION

It is an object of the present invention to provide an efficient logging scheme that can be used to recover a transaction processing system after a failure occurs.

It is another object of the present invention to provide a logging scheme where parallel operations are possible.

The foregoing objects and other objects are achieved in the present invention using a differential logging method that allows commutative and associative recovery operations. The method includes the steps of taking a before-image of database in main memory before an update to the database; taking an after-image of the database after the update; generating a log by applying bit-wise exclusive-OR (XOR) between the before-image and the after-image; and performing either a redo or undo operation by applying XOR between said one or more logs and the database.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a system diagram of conventional recovery architecture of a main-memory database management system storing the database entirely in main memory.

FIG. 2 is a flow chart of a restart process used in the conventional recovery architecture in the case of system crash.

FIGs. 3a and 3b are flow charts of a two-pass log-play process used in the conventional

recovery architecture during a restart process.

FIG. 4 is a diagram of the structure of update log records used in the present invention to capture changes in the database.

FIG. 5 is a diagram illustrating the redo and undo operations applied to update log records of the present invention during a restart process.

FIGs. 6a to 6e are an illustration comparing the differential logging scheme of the present invention with the physical logging scheme of the conventional art.

FIGs. 7a and 7b flow charts of a one-pass log-play process used in the present invention to recover the database from a backup made with a consistent checkpointing scheme.

FIGs. 8a and 8b are flow charts of a fuzzy checkpointing process used in the present invention to make a backup without blocking other transactions.

FIG. 9 is a flow chart of an update process used in the present invention in combination with the fuzzy checkpointing process.

FIGs. 10a and 10b are flow charts of a modified two-pass log-play process used in the present invention to recover the database from a backup made by the fuzzy checkpointing process.

FIGs. 11a and 11b are flow charts of a modified one-pass log-play process used in the present invention to recover the database from a backup made by the fuzzy checkpointing process.

FIG. 12 a diagram of one embodiment of a logging architecture of the present invention where multiple log disks are used to distribute log records and to enable parallel logging.

FIG. 13 is a diagram of one embodiment of a restart architecture of the present invention where multiple log disks and backup disks are used to enable parallel restarting.

FIG. 14 is a flow chart of a fully parallel restart process used in the present invention to recover the database as quickly as possible using multiple log disks and backup disks.

FIG. 15 is a diagram of one embodiment of the backup loader (BL) module used in the fully parallel restart process of the present invention.

FIG. 16 is a diagram of one embodiment of the log loader (LL) module used in the fully parallel restart process of the present invention when a disk can transfer log records faster than a single CPU can handle.

FIG. 17 is a diagram of another embodiment of the LL module used in the fully parallel restart process of the present invention when a disk can transfer log records faster than a single

CPU can handle.

DETAILED DESCRIPTION OF THE INVENTION

FIG. 4 shows the structure of update log records and database pages used in the present invention. A storage structure is assumed where a database consists of multiple pages of a fixed size, with each page having many slots. A different storage structure may need a modification, but such modification should be apparent to those skilled in the art. A preferred embodiment of the present invention uses at least five types of log records: transaction start, transaction end, update, checkpointing start, and checkpointing end. The first two are used to record the beginning and the end of a transaction while the last two are used to record the beginning and the end of a checkpointing process.

An update log record used to store changes in the database comprises of a log header and a log body. FIG. 4 shows a variety of fields in the log header according to a preferred embodiment. The "LSN (Log Sequence Number)" field stores the identity of the current log record by preferably storing the physical address of log record on disk. The "TID (Transaction ID)" field stores the identity of the transaction associated with the current log record. The "PrevLSN" field stores the identity of the log record that was most recently created by the same transaction so that the information can be conveniently used to chain the log records of a transaction for fast backward retrieval. The "Type" filed stores the type of log record. The "Backup ID" field stores the relation between the log record and the changed page. This field needs to be maintained only when using a fuzzy checkpointing scheme, which will be explained later in FIGs. 8a and 8b. The "Page ID" field stores the identity of a page where the update occurred. The "Offset" field stores the identity of a slot inside a page where the update occurred. The "Size" field stores the length of the updated slot.

The body of a log record stores the differential log information. Specifically, it stores the bit-wise exclusive-OR (XOR) result of the data image of the database before update ("before image") and the data image of the database after update ("after image"). This differential logging scheme storing only the differentials is distinguished from the conventional physical logging scheme where both the before image and the after image are stored. For example, if the before image is "0011" and the after image is "0101", the present invention stores only differential, namely, the XOR result of "0110".

FIG. 5 illustrates the redo and undo operations applied to the update log records of the present invention during a restart process. First, the log header is consulted to find the location in the database, associated with the current log record. In a preferred embodiment, the location is

determined from the "Page ID" and "Slot ID" information found in the log header. The redo or undo operation is performed by applying bit-wise XOR between the data found in the location and the log body of the current log record and writing the XOR result into the location. Unlike the conventional physical logging scheme, there is no difference between the redo operation and the undo operation in the present invention.

FIGs. 6a to 6e shows a comparison between the differential logging scheme of the present invention and the physical logging scheme of the conventional art. Since the operations in the conventional physical logging scheme are not commutative and associative, the redo operations, for example, must be performed in the sequential order of log record creation.

Consider a situation where a system crash occurs after processing three transactions, T1, T2, and T3, as in FIG. 6a. Suppose these transactions change the four-byte integer in a slot of the database in the order of 0, 7, 6 and 14. FIGs. 6b and 6c show a sequence of log records created as the slot image changes when the conventional logging scheme and the differential logging scheme are used, respectively. Whereas the body fields of the conventional log records contain both the before and after image, those of differential log records contain only the difference between the before and the after image.

Upon the system crash, if redo operations were applied in the same sequence as the original sequence, a correct recovery would result for both logging schemes. FIG. 6d shows that both logging schemes recover the correct value of 14 by applying redo operations in the order of log record creation. Note that the redo operation of the conventional logging scheme copies the after image into the corresponding slot of the database.

However, if redo operations were done in a different sequence from the original sequence, a correct recovery would not be possible in the conventional logging scheme. In contrast, the differential logging scheme of the present invention enables an accurate reconstruction regardless of the order of applying the log records. FIG. 6e shows the consequence of different orders of applying the redo operations. With the differential logging scheme, a correct recovery results regardless of the order of applying the redo operations. This differential logging scheme may be applied to both a main-memory DBMS where the database is primarily stored in main memory and a disk-resident DBMS where the database is primarily stored in disks.

FIGs. 7a and 7b is a flow charts of a one-pass log-play process used in the present invention to recover the database from a backup made using a consistent checkpointing scheme. There are two categories of consistent checkpointing schemes: transaction consistent and action consistent. "Transaction consistent" means that no update transaction is in progress during

checkpointing. In other words, a transaction-consistent checkpointing process can start only after all the on-going update transactions are completed, and no update transaction can start until the checkpointing process completes. "Action consistent" means that no update action is in progress during the checkpointing process. When using a consistent checkpointing scheme, the two-pass log-play process of FIGs. 3a and 3b can be used directly with the differential logging scheme of the present invention.

One benefit of the present invention is to make it possible to complete the log-play process by scanning the log only once. In the one-pass log-play process of the present invention, the log records are scanned in the opposite direction to the log record creation sequence, i.e., from the end of the log. When scanning the log backward, a transaction end log record is encountered before any other log records of the transaction. When an aborted transaction is encountered, there is no need to play the records of the transaction. In other words, once the committed transactions are identified, only the records of the committed transaction can be played, skipping the records of the aborted transactions.

Since nothing needs to be done for a loser transaction, it is treated as the same as an aborted transaction. As mentioned above, in the conventional methods, redo operations must be performed for all the transactions in the sequence of log record creation. Redo operations may be skipped for a loser transaction, but since one cannot determine whether a transaction is a loser transaction or not in advance, redo operations are done even for those would-be loser transactions. Therefore, undo operations are needed for those loser transactions after they are identified.

FIGs. 8a and 8b is a flow charts of a fuzzy checkpointing process used in the present invention to make a backup without blocking other transactions. "Fuzzy checkpointing" means that an update transaction and the checkpointing process may proceed in parallel. Fuzzy checkpointing is often preferred to consistent checkpointing because it allows other transactions to proceed during the checkpointing period.

When using fuzzy checkpointing together with the differential logging scheme of the present invention, two synchronization problems must be dealt with for correct recovery of the database. First, although an update transaction and a checkpointing process may occur in parallel as long as the two apply to different pages, a database page should not be backed up during checkpointing while the same page is being updated by a transaction. Otherwise, a mixture of both the before and the after images are copied, making it difficult to correctly recover from a crash. To handle the first problem, the present invention provides a synchronization mechanism so that the process of backing up a page and the process of updating a page occur in a locked

state (as an atomic unit) (808 and 811).

Second, a mechanism is needed to determine whether a backed-up page reflects the database after a log record creation or before a log record creation. If the backup was made after a log record creation, there is no need to play the log record because the log record already reflects the changes. Since the present invention uses XOR for redo and undo operations, a mechanism is necessary to determine whether to play a log record or not. To deal with the second problem, the present invention maintains a field storing the most recent backup identifier in each page (809) and copies it into log records.

FIG. 9 is a flow chart of an update process used in the present invention in combination with the fuzzy checkpointing process of FIGs. 8a and 8b. To handle the two above-mentioned synchronization problems, a page is updated in a locked state (901 and 907), and the backup ID stored in a page is copied into an update log record (903). In order to update a page, the latch for the page is first acquired to enter into a locked state (901). Then, an update log record is created before updating (902), and the backup ID in the page header is copied into the log record (903). Then, the page is updated (904), and all the dirty flags in the page header are set (905). As mentioned above, there are two dirty flags: one for each of the two backups. Finally, the log record is appended to the log buffer (906), and the latch is released (907).

FIGs. 10a and 10b is a flow charts of a modified two-pass log-play process used in the present invention to recover the database from a backup made by the fuzzy checkpointing process. The difference from the two-pass log-play process of FIGs. 3a and 3b is that a special action is taken for update log records located between a checkpointing start record and the matching checkpointing end record. For such a log record, the backup ID in the record is compared to the backup ID stored in the page to which the log record refers. If the two IDs are same (1002), it indicates that the log record was created after the page was backed up. Since the page does not reflect the change by the log record, the log records needs to be played. But, if two IDs are different, it indicates that the log record was created before the page was backed up. Since the page already has the change corresponding to the log record, there is no need to play the log record. After encountering the checkpointing end record, the same steps as FIGs. 3a and 3b are taken (1001).

FIGs. 11a and 11b is a flow chart of a modified one-pass log-play process used in the present invention to recover the database from a backup made by the fuzzy checkpointing process. The difference from the one-pass log-play process of FIG. 7 is that a special action is taken for update log records located between a checkpointing end record and the matching checkpointing start record. For such a log record, the backup ID in the record is compared to the

backup ID stored in the page to which the log record refers. If the two IDs are same and the corresponding transaction was committed, the record should be played to redo the change by the record. If the two IDs are different and the corresponding transaction was aborted, the record should be played to undo the change by the record. Otherwise, the record is just skipped.

FIG. 12 shows an embodiment of a logging architecture of the present invention where multiple log disks are used to distribute log records and to enable parallel logging. Unlike the physical logging scheme, the differential logging scheme allows us to freely distribute log records to multiple disks to improve the logging performance because the commutativity and associativity of XOR used in the differential logging scheme enables processing of log records in an arbitrary order. In this embodiment, log records are distributed to multiple log disks based on transaction identifiers (TIDs). There is a different log manager (1201) for each log disk (1203). When a transaction starts, the identity of the log manager (LogMgr ID) having the least amount of load is assigned to the transaction so that the log records generated by the transaction are always stored in the log disk.

FIG. 13 shows an embodiment of a restart architecture of the present invention where multiple log disks and backup disks are used to enable parallel restarting. As mentioned in FIG. 2, the restart process comprises the four sub-processes, BR, BP, LR, and LP. In this embodiment, three types of parallelism are utilized to speed up the restart process. First, a different backup loader (BL) (1302) is instantiated for each backup disk (1307), and it performs the BR process (1302) and the BP process (1301) in a pipeline manner. Each BL runs independently of other BLs. Second, a different log loader (LL) (1305) is instantiated for each log disk (1308), and it performs the LR process (1306) and the LP process (1304) in a pipeline manner. Each LL runs independently of other LLs. Third, even BLs and LLs are run concurrently.

FIG. 14 is a flow chart of a fully parallel restart process used in the present invention to recover the database as quickly as possible using multiple log disks and backup disks. First, all the pages in the database are filled with 0s (1401). Since 0 is the identity of the XOR operation, this allows the BP process to use the XOR operation to restore a page in the database instead of copying a page into the database. Second, one BL module and one LL module are created for each backup disk and log disk (1402 and 1403), respectively. They run independently and in parallel.

FIG. 15 is a diagram of one embodiment of the backup loader (BL) module used in the fully parallel restart process of the present invention. The BL runs the BR process (1504) and the BP process (1502) in a pipeline manner or in the producer/consumer way. The BR process reads the backup disk (1505) page by page and appends each page to the page buffer (1503), which is

used like a pipeline. The BP process takes a page from the buffer and applies it to the matching page in the database (1501). A novel aspect of the present invention is that the BP process uses the XOR operation when applying a backed-up page to the database. This allows us to run BLs and LLs concurrently, which was not possible in the conventional art.

FIG. 16 is a diagram of one embodiment of the log loader (LL) module used in the fully parallel restart process of the present invention when a disk can transfer log records faster than a single CPU can handle. In this embodiment, multiple LP processes (1601) are executed concurrently to utilize multiple CPUs. In addition to the LR and LP processes, the LL module uses another process (1604) to dispatch log records to multiple LP processes such that all the log records of a transaction are assigned to the same LP process. Then, each LP process can maintain transaction tables (1601) such as the LTT used in a two-pass log-play process and the CTT and the ATT used in a one-pass log-play process locally.

FIG. 17 is a diagram of another embodiment of the LL module used in the fully parallel restart process of the present invention when a disk can transfer log records faster than a single CPU can handle. In the previous embodiment of FIG.16, the dispatcher process (1604) can be a bottleneck as the number of concurrent LP processes increases. Another problem is the overhead of acquiring and releasing locks for log record queues (1603) to use them concurrently. To deal with these problems, LP processes (1702) access the log buffer (1703) directly in the current embodiment and partition log records based on their transaction identifiers. A simple scheme is to use the modulo operation. For example, when there are three LP processes, the first process takes the log records with the transaction identifiers that have the remainder of 0 when divided by 3.

In this embodiment, each log buffer page has a counter. This counter is reset when the LR process (1704) fills it with the data read from the log disk (1705). When an LP process finishes scanning a buffer page, it increment the counter of the page in a locked state. Then, when the counter has the same value as the number of LP processes, the buffer can be flushed.

While the invention has been described with reference to preferred embodiments, it is not intended to be limited to those embodiments. It will be appreciated by those of ordinary skilled in the art that many modifications can be made to the structure and form of the described embodiments without departing from the spirit and scope of this invention.